IAnticorruption – A Domain-Driven Design Approach To More Robust Integration

Sam Peng
speng@customhouse.com

Ying Hu
yhu@customhouse.com

ABSTRACT

Custom House's new currency exchange system is integrated with a legacy system. After a few years of growth, the two systems were so intricately tangled that even small changes made in the integration layer would have unpredictable side effects. Refactoring on the integration layer was risky and time consuming. The situation called for a revolutionary redesign.

The solution was to introduce an anticorruption layer to isolate the two systems. This layer encapsulated the translation of conceptual objects and actions between the two systems, insulating the domain layer from knowing the existence of the other system. By freeing the domain layer from performing tasks that were only relevant to the other system, the anticorruption layer allowed additional external systems to be integrated without requiring any changes to the domain layer itself. Full implementation of an anticorruption layer reduced overhead of legacy integration from 30% of total development to 10%.

The biggest challenge in implementing the anticorruption layer is to control the complexity of translation work. This was managed in an innovative way: by building an object model reflecting the implicit model of the legacy system. Our experiences show that an external system need not be object-oriented for its model to be adequately abstracted, and this has proven to be the key to a clean and extensible translation.

Keywords

Domain-Driven design, anti-corruption layer, domain model, integration, observer pattern

BACKGROUND

Like most enterprise software applications, Custom House's new currency exchange system needs to integrate with a legacy back office system to fully support business workflow. The new system (SPOT) handles most front-end online transactions. When a currency exchange transaction is booked through SPOT, information needs to be sent to the back office for further processing, which is handled by a legacy system (TBS).

While SPOT is a multi-tier Microsoft .NET enterprise application built with object-oriented technologies, TBS is a Microsoft FoxPro application based on tables and records.

INTEGRATION

To centralize data conversion logic and cross-platform communication, we built a component

called TBSExport as a gateway between SPOT and TBS.

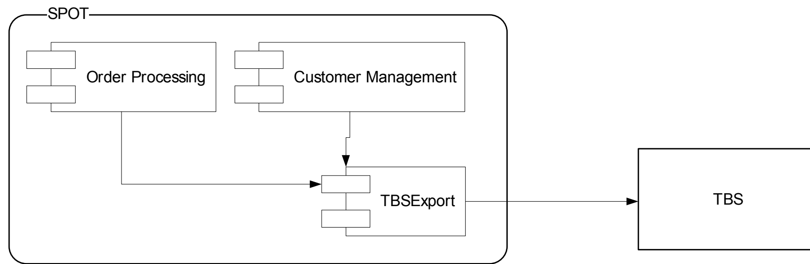

Figure 1. Dependency between SPOT and Tbs in old design

(Among the numerous types of information SPOT sent to TBS, we show only two of the most important types here, Order and Customer, to simplify the discussion.)

The TBSExport component would take generic data structures as input (in the form of .NET DataSets), convert them to a TBS-recognizable format, then output them to TBS. For example, when an order was booked, SPOT would build a dataset containing information about the booked order, then call TBSExport:

```
public class OrderManager
{
   public void BookOrder(Order order)
   {
            //...book order and update order entry in database
            DataSet orderDataSet = OrderDataSetBuilder.BuildTbsDataSet(order);
            TbsExport.ExportOrder(orderDataSet);
   }
}
```

(code example 1)

TBSExport, on receiving this request, would delegate to one of several concrete exporters for processing and communication.

CORRUPTION

The initial design worked for a while. But as both SPOT and TBS evolved to support new features, integration became a heavy burden. Developers found translation between the two systems hard to understand. Two years after the first release of SPOT, tasks related to TBSExport were consuming nearly 30% of our total development effort! Our QA team considered it one of the most buggy and brittle areas in the system. Finally, one project was deemed hopeless due to a significant TBSExport enhancement requirement. Both managers and developers screamed for a revolutionary redesign.

So what went wrong?

1. SPOT domain was highly coupled to TBS

From code example 1, we see that SPOT does two things when sending information to TBS:

1. Build a dataset containing specific data

2. Call an exporting service defined in TBSExport component

In step 1, SPOT domain objects converted themselves into datasets. Because the conversion code was littered throughout the domain objects themselves, it obscured the real business logic of those objects and made them much bulkier than they should have been. Later, we isolated most of the conversion code in separate classes, such as OrderDataSetBuilder. This did save domain objects from being swamped, but the domain layer was still sprinkled with converters that contained no SPOT business logic.

In step 2, it was SPOT's responsibility to initiate exporting. This required a reference to TBSExport in almost all major SPOT operations, such as customer creation, order booking, payment releasing etc. At first, this dependency only existed in a service layer that handled workflow. But before long, it crept into domain layer too, causing SPOT domain objects to be modified to contain TBS specific knowledge. The "BankDeposit" class, for instance, had an internal member "TBSFile" – a type defined inside the TBSExport component.

```
public class BankDeposit
{
        TBSFile tbsFile;
        public void DoDeposit()
        {
            // domain logic for deposits…
            tbsFile.Export();
        }
}
```

Such coupling between SPOT and TBSExport caused maintenance headaches.

- SPOT domain logic was difficult to test in isolation.
- Unit tests for TBSExport required complicated setup to prepare SPOT objects.
- Bugs related to exporting logic were time-consuming to track and fix. A simple change in TBS could cause unexpected issues in SPOT and vice versa.

2. Translation was done at a low level of primitive data types

TBSExport exposed interfaces that accepted datasets. These datasets were like a flattened version of SPOT objects. OrderDataSet, for example, contained bits and pieces of information extracted from SPOT Order object. Upon receiving datasets, TBSExport needed to further interpret and process them to produce data understandable to TBS.

The translation logic was so complicated because SPOT and TBS are based on two very different models. In SPOT, an "Order" is an aggregate containing one to many "Line Items". Each "Line Item" is associated with one to many "Contracts" by a "Drawdown" object. When exporting an order to TBS, information about the order, line items, drawdowns and contracts were all bundled into an OrderDataSet.

TBS, by contrast, uses a flat table structure to represent different types of “currency exchange deal”. Each TBS deal is one record in the table. The SPOT concepts don’t exist in TBS, so mapping logic would convert a SPOT contract into two deal records, a SPOT line item into one deal record, and a SPOT drawdown into two deal records. A SPOT order was indirectly mapped to multiple records.

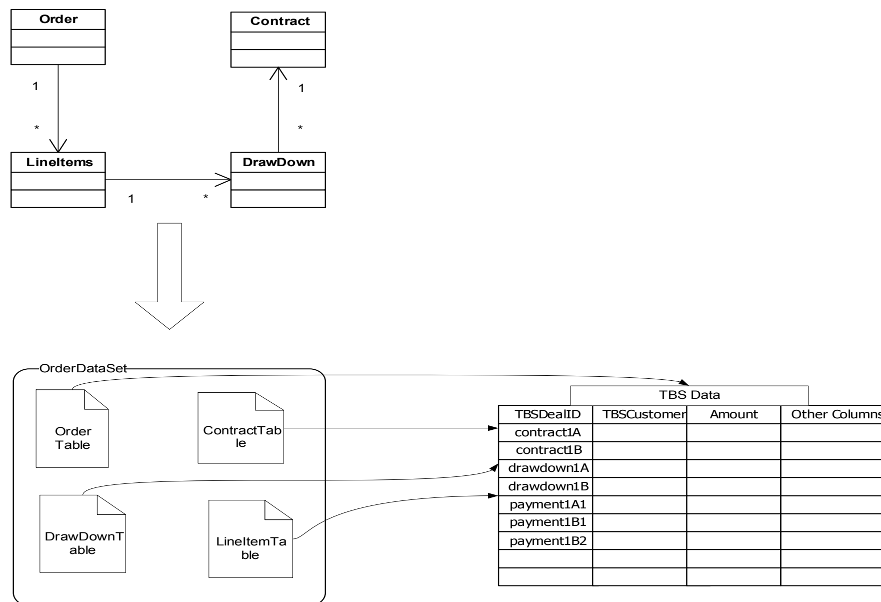

Figure 2. Old translation

Both OrderDataSet and the TBS deal table were structures containing only primitive datatypes, like strings and integers. Translation logic had to understand their meaning in both contexts, and figure out the intricate associations between them. Such a low level field-to-field mapping was laborious and error prone, containing numerous details and subtleties, and duplicating logic across the two systems [3]. The mapping logic in OrderExporter alone had over 3000 lines of code. This complexity was one of the worst sources of confusion that made the component unmanageable.

3. Business logic was tangled with technical details in TBSExport

The TBSExport component contained a group of Exporter classes at its core:

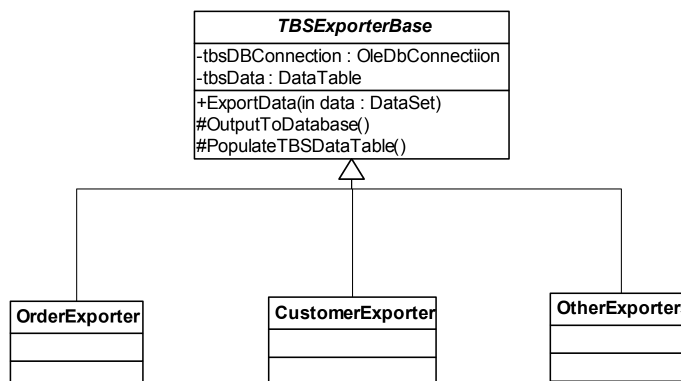

Figure 3. Old TbsExport design

The abstract base class, TBSExporterBase, provided implementations to physically create and save output data files. Each subclass had to override "PopulateTBSDataTable()" to do specific translation and "OutputToDatabase" to perform corresponding database operations. These two operations were of very different types. One handled business logic (making TBS deal records) while the other was purely technical (saving those records to disk). Here is an example of how these mixed responsibilities complicated the class itself and fragmented business logic in data rows, data tables and database connections:

```
public class CustomerExporter
{
   protected override void PopulateTBSDataTable(DataSet dataSet)
   {
            DataRow drSpotCustomer = dataSet.Tables[0].Rows[0];
            DataRow drTBSCustomer = tbsData.NewRow();

            drTBSCustomer[COMPANY] = drSpotCustomer["CompanyName"];
            //......more
            tbsData.Rows.Add(drTBSCustomer);
   }
        protected override void OutputToDatabase()
        {
            OleDbCommand dbCommand=new OleDbCommand(tbsDBConnection);
            //...
            foreach (DataRow dataRow in tbsData.Rows)
            {
                  dbCommand.CommandText= BuildQueryString();
                  dbCommand.Parameters[COMPANY].Value= dataRow [COMPANY];
                  //...more
                  dbCommand.ExecuteNonQuery();
            }
        }
}
```

REDESIGN - CREATING AN ANTICORRUPTION LAYER

TBSExport was so intricately tangled with the SPOT domain that when it needed to be significantly extended, developers would dread touching that part of the code. In December 2005, the team decided that incremental refactoring would not be sufficient because unpredictable side effects made even small changes risky and time-consuming. The integration layer between SPOT and TBS needed major surgery, and we decided we would rather bite the bullet. With help from Eric Evans and the Domain Language team, our entire team – project managers, developers and QA – started to work on TBSExport redesign. We decided on the following strategy:

- Implement a new TBSExport gateway component as a spike. It must be self-contained and stand-alone. Make sure this new component is well designed and thoroughly unit

tested;

- Create a mechanism to connect this component to SPOT, but in a loosely coupled fashion (so SPOT won't know about it);
- When satisfied that the new mechanism is behaving properly, switch the new component "on" inside SPOT, and do integration and regression testing;
- Finally, delete all the old TBS exporting code, leaving only the new, decoupled design.

This plan enabled us to build an entirely new component from a clean slate without breaking existing functionality, so that our system wouldn't be in an unworkable state for very long. It also enabled us to run the old and new code side-by-side until we were satisfied that the new component was ready.

The most important design decision was to make TBSExport a true **anticorruption layer** standing between SPOT and TBS. An anticorruption layer "*is not a mechanism for sending messages to another system. Rather, it is a mechanism that translates conceptual objects and actions from one model and protocol to another.*" [2] In other words, we wanted our new component to be an isolating layer that consumed SPOT domain objects directly – leaving SPOT itself blissfully ignorant of any translation work or any model besides its own – and completely encapsulate translation logic. To achieve this, we did the following:

1. Define a new TBSExport façade interface that fits into the SPOT model

Compare the two interfaces:

Before	*public void ExportOrder(DataSet orderDataSet);*
After	*public void OnOrderBooked(Order order);*

The old interface required SPOT to convert its objects into a DataSet. The new interface simply accepts the SPOT domain object as a parameter. Thus SPOT is only required to implement interfaces in its own language, rather than to do something foreign such as "export this dataset".

2. Discover the TBS model and re-abstract TBS's behavior

The legacy TBS system is not object-oriented by nature. This by no means implies that it doesn't have a model, just that the model is buried in various data records and hinted at in procedures. We found it necessary to discover TBS's model in order to explicitly express the semantics on TBS side. In our Order export example, even though there is no explicit TBS deal record to match a SPOT order, a conceptual order that links a series of TBS objects does exist in TBS:

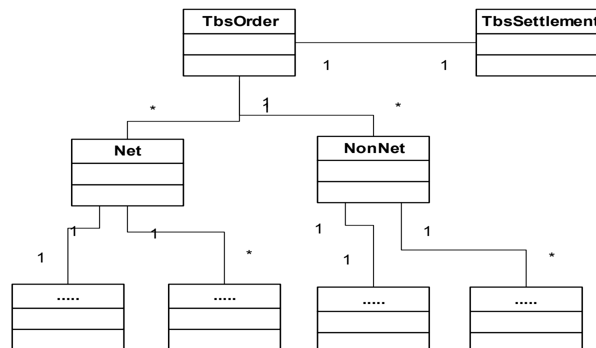

Figure 4. Our representation of the implicit TBS model

Once we had written code to express the TBS model, we could translate a SPOT order into its

equivalent on the TBS side:

```
public class TbsOrderTranslator
{
    public TbsOrder TranslateSpotOrder(Order spotOrder)
        {
                TbsOrder tbsOrder = new TbsOrder();
                tbsOrder.Customer = MakeTbsCustomerId(spotOrder.CustomerId);
                tbsOrder.Branch = spotOrder.Branch.BranchCode;
                //....more
                tbsOrder.Settlement = ComputeSettlement(tbsOrder);
                return tbsOrder;
        }
}
```

Having such a "TbsOrder" object in TBSExport was a breakthrough. It became a key to understanding the associations among a group of TBS objects. It enabled us to translate between two object models rather than manipulating primitive data, taking full advantage of each model's ability to represent its data meaningfully. Now we were able to speak to each model in its own language.

A subsequent step then maps TBSOrder to TBS records. *This is a straightforward, mechanical mapping, involving no complex business logic.* All that had been taken care of when translating from the SPOT to the TBS model..

3. Separate TBS platform-specific communication details from object translation logic.

Unlike the old Exporter classes that mixed translation with communication, we built the new TBSExport component in three layers, each of which has one and only one responsibility:

- *Translators* translate SPOT objects to TBS objects;
- *Record generators* build TBS data records out of those objects;
- *File writers* output those records to physical dbf files.

Here's an example of how a SPOT order is put through each of those layers in turn, resulting in a complete TBS export:

```
public void OnOrderBooked(Order order)
{
    //1) Translate Spot Order to TBS Order:
    TbsOrder tbsOrder = new SpotToTbsOrderTranslator(order).TranslateSpotOrder();

    //2) Create TBS specific data structure from TBSOrder:
    TbsTable tqrTable = new OrderTqrGenerator(tbsOrder, database).GenerateOrderTqrTable();

    //3) Write TBS Files
    GetTbsFileWriter(tqrTable).Write();
}
```

The entire TBSExport component structure is shown below:

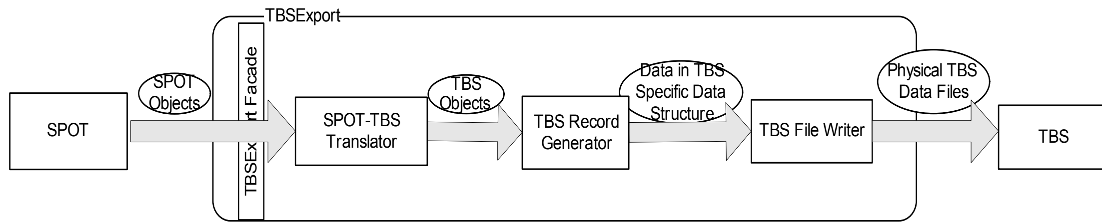

Figure 5. New TBSExport design

Among the many benefits of this separation, it is easy to unit-test each stage of the export process. When bugs occur, we pinpoint them quickly.

4. Invert dependency from SPOT components to TBSExport.

In the old design, it was SPOT's responsibility to push information into TBSExport. This demanded that SPOT know when and how to call TBSExport, resulting in many SPOT objects holding reference to TBSExport and knowing a great deal about TBS. This "push" approach was problematic because it burdened SPOT – which is already a complex system containing much business logic – with responsibility for external systems not core to its domain. Furthermore, there were still other external systems needing integration with SPOT in the near future, and were this to be implemented in the same fashion, SPOT would be overwhelmed.

A better alternative is the observer pattern approach – with TBSExport being notified when certain events happen in SPOT. We implemented the observer pattern using .NET events.. Our implementation was made simpler by a particular feature that allows events to be defined "static", at class level rather than instance level. This technique enabled us to hook up SPOT events with TBSExport handlers in a single centralized location, at server startup time, long before any object instances are created. It also gave us the flexibility to configure test projects differently.

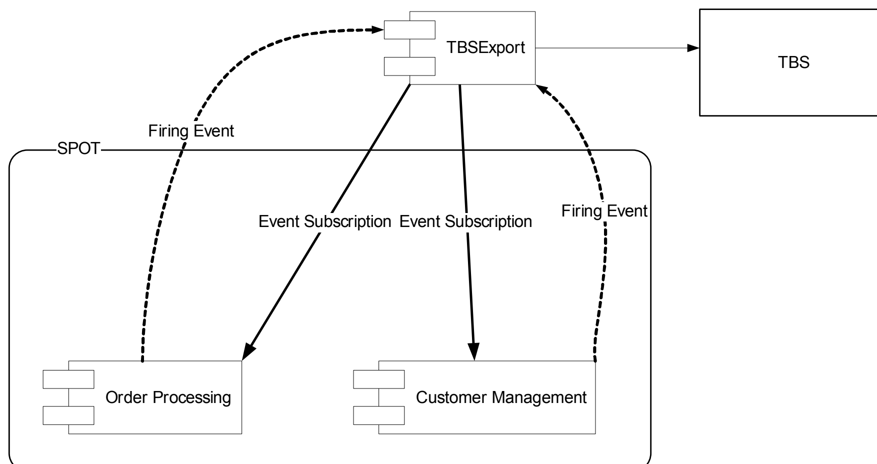

Figure 6. Dependency between SPOT and Tbs in new design

For example, OrderManager has a static event that is triggered when any order object is booked:

```
public class OrderManager
{
    public static event OrderEventHandler OrderBooked;
    public void BookOrder(Order order)
    {
        //...book order and update order entry in database
        If (OrderBooked != null)
            OrderBooked(order);         //fire event
    }
}
```

Unbeknownst to SPOT, TBSExport subscribes to this OrderBooked event:

```
public class TbsExportManager
{
    public void SubscribeToSpotEvents()
    {
        OrderManager. OrderBooked += new OrderEventHandler(OnOrderBooked);
        //subscribe to other SPOT events
    }
}
```

When an order is booked, OrderManager simply raises the OrderBooked event. *It no longer has any knowledge of the consequences of this outside its own sphere.* When TBSExport, which subscribes to this event, gets notified, its own OnOrderBooked() method is called, and the export process takes place. This design inverted the dependency, freeing SPOT from knowing anything about external systems. *It also allowed additional external systems to receive the information they needed from SPOT, without requiring any changes to SPOT itself.*

All the above design changes led to the point where we could remove every single reference to TBS from the SPOT domain layer.

CONCLUSION

The original design to create the TBSExport component as an integration gateway was a common practice. It worked ok at the beginning. However, as the pair of systems evolves, not only did the complexity of TBSExport grew out of control, the gateway also led TBS knowledge into SPOT domain layer, burdening it with responsibilities beyond its own sphere. The key cause is that the gateway didn't completely encapsulate the translation logic between SPOT and TBS, resulting two very different conceptual models crossing each other's boundary.

The solution is to re-design the TBSExport as a true anticorruption layer, isolating SPOT from TBS such that TBS knowledge doesn't leak into the SPOT business domain and vice versa. Our implementation of such layer contains the following key items:

- TBSExport provides services that speak SPOT domain language. Its interfaces to SPOT take domain objects like Order and Customer.
- TBSExport completely encapsulates translation logic between SPOT objects to TBS data records. The translation complexity was managed in an innovative way: by first abstracting the implicit model of TBS, not in its entirety but just to the extent required for translation. Our experiences show that an external system need not be object-oriented for its model to be adequately discovered and expressed, and this has proven to be the key to a clean and understandable translation.
- Translation logic is separated from low-level communication details.
- SPOT domain objects don't reference and depend on TBSExport; TBSExport is loosely coupled to the SPOT domain through an observer pattern.

After the redesign, the TBSExport component has become a complex piece of software in its own right, but in a well-encapsulated and decoupled form that is easier to maintain. The remodeling project took a 4-6 team 6 weeks to finish. Yet the effort was soon paid off. Subsequent projects report less than 10% of total effort on TBSExport (66% less than the original), and the days when developers feared to approach that part of the system are long past.

ACKNOWLEDGEMENT

We would like to thank Heather Regehr for the support from management; Eric Evans for his insights and inspiration; Daniel Gackle for his contribution to both the project and this paper, Brenda Lowe and our QA team for the excellent team work, Taj Khattra, Alex Aizikovsky, George Zhu, Todd Ariss and the entire Custom House development team for their effort that made the project successful.

REFERENCE

[1] Eric Evans, Domain-Driven Design, Tackling Complexity in the Heart of Software, Addison-Wesley, 2003, ISBN 0-321-12521-5.

[2] Ying Hu and Sam Peng, So We Thought We Knew Money, available from http://www.domaindrivendesign.org/practitioner_reports/hu_ying_2007_01.html